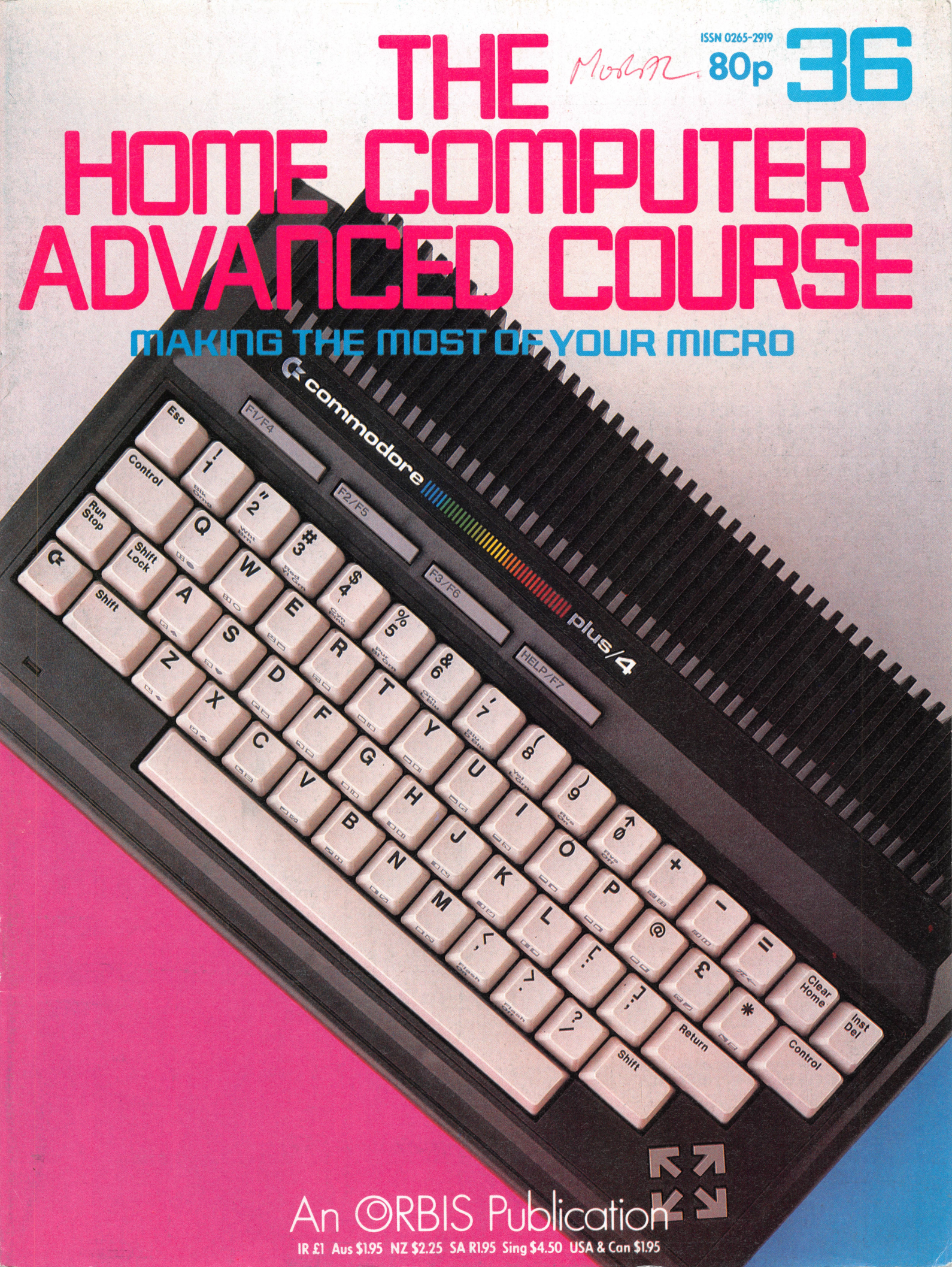
ISSN 0265-2919

80p

36

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO



An ORBIS Publication

IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

Next Week

- **Following our examination of the state of the home computer market, we cast a similar eye over peripherals — everything from infrared joysticks to double disk drives**
- **Our Workshop series brings you a discussion of real-time programming and some test programs for the digital-to-analogue converter**
- **In BASIC programming we extend the search utility's scope by 'bolting on' a replace facility**
- **Robotics depends on making decision-making mobile: we investigate intelligent robotic movement**

QUIZ

1) Where do you find 'demons'?
2) What does 'REPLICATE' do?
3) What is 'continuous path training'?
4) What is a hacker?

Answers To Last Week's Quiz

1) When a file update program is run, the oldest version of the file — known as the 'grandfather' file is deleted, and is replaced by what was previously the 'father' file.

2) Decoders translate digital instructions from the computer to its peripherals into electrical signals, and vice versa.

3) Transducers take a measurement in one form — such as pressure — and convert, or transduce, it into another form — such as voltage.

4) The system variable 'PAGE' is part of the BBC Micro operating system. It contains the address of the start of the BASIC text area.

Editor Mike Wesley; **Art Director** David Whelan; **Technical Editor** Brian Morris; **Production Editor** Catherine Cardwell; **Art Editor** Claudia Zeff; **Chief Sub Editor** Robert Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Editorial Assistant** Stephen Malone; **Sub Editor** Steve Mann; **Researcher** Melanie Davis; **Staff Writer** Steve Colwill; **Contributors** Geoff Bains, Harvey Mellar, Mike Curtis, Steve Colwill, Chris Naylor, Tony Harrington, Jim Lennox, Steve Malone, Ted Ball; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Controller** Peter Taylor-Medhurst; **Circulation Director** David Breed; **Marketing Director** Michael Joyce; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders. **UK/EIRE** – Price: 80p/IR£1. Subscription: 6 months: £23.92. 1 Year: £47.84. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Price: 80p. Subscription: 6 months air: £37.96. Surface: £31.46. 1 year air: £75.92. Surface: £62.92. Binder: £5.00. Airmail: £5.00. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Price: 80p. Subscription: 6 months air: £43.94. Surface: £31.46. 1 year air: £87.88. Surface: £62.92. Binder: £5.00. Airmail: £8.31. **AMERICAS/ASIA/AFRICA** – Price: US/CAN$1.95/80p. Subscription: 6 months air: £51.74. Surface: £31.46. 1 year air: £103.48. Surface: £62.92. Binder: £5.00. Airmail: £9.44. **SOUTH AFRICA** – Price: SA R1.95. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Price: 80p. Subscription: 6 months air: £55.38. Surface: £31.46. 1 year air £110.76. Surface: £62.92. Binder: £5.00. Airmail: £9.84. **AUSTRALIA** – Price: AusS1.95. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Price: NZS2.25. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.

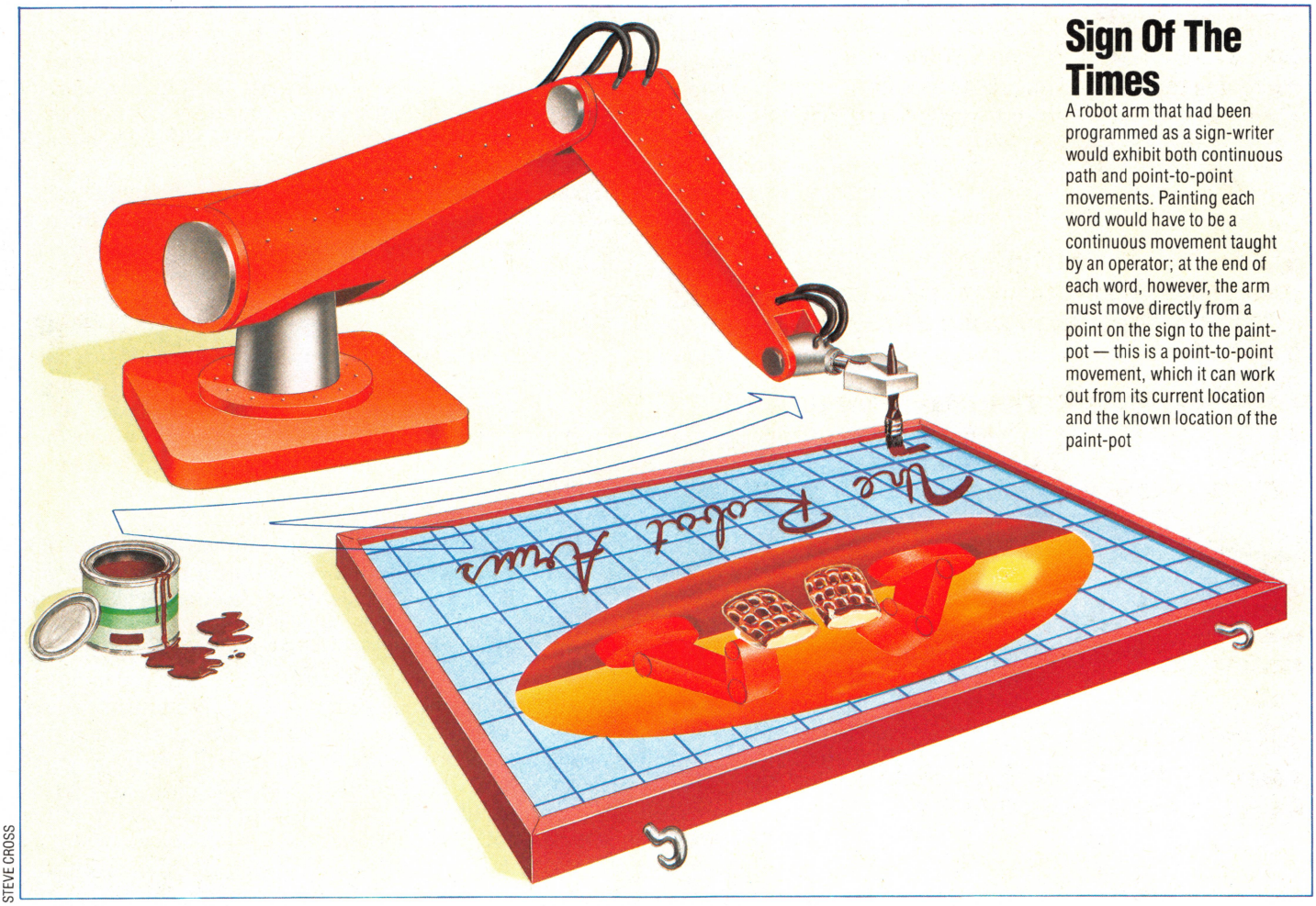
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 6711. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.

NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.

ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited, Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY IAN McKINNELL

CALCULATED MOVES



STEVE CROSS

Sign Of The Times

A robot arm that had been programmed as a sign-writer would exhibit both continuous path and point-to-point movements. Painting each word would have to be a continuous movement taught by an operator; at the end of each word, however, the arm must move directly from a point on the sign to the paint-pot — this is a point-to-point movement, which it can work out from its current location and the known location of the paint-pot

Having considered methods of robot movement, and the design of robot 'arms' and 'hands', we now move on to discuss the various ways in which a robot arm may be programmed to carry out seemingly 'intelligent' tasks.

We have already seen how robot arms may be constructed in such a way as to resemble a human limb — they have 'skeletons' to provide structure, and 'muscles' to provide motive power. But the arm still needs 'intelligence' to carry out tasks.

The idea of an intelligent arm may at first seem nonsensical. However, the form of intelligence we are considering here is not the kind of high-level intelligence possessed by humans, but something considerably less complex. Let's consider a simple human action. You are sitting at a table that is empty apart from a small object situated on its left-hand side. Your task is simply to move this object from the left to the right of the table top. Two forms of intelligence are involved here. The first involves the perception of both the table and the object and the decision to move the object from one side to another. This involves 'aware thought', and it is tied up with concepts such as 'intention' and 'goal-orientated behaviour'. The intelligence we need to consider is the much lower-level form that is needed to move your arm and hand correctly *after* you have decided on the task that needs to be carried out — to move your hand to the right position and to ensure that your hand grasps the object and releases it at the correct time.

HUMAN TRAINING

This may seem both easy and obvious — but if you doubt that this is in fact an intelligent act, just watch a small child attempting to follow the same sequence. The infant will often fail to pick up the object, will move it to an inappropriate position and will, generally, appear quite uncertain as to what is required. The child is attempting to acquire the intelligence necessary to move its arms and

hands around in an unfamiliar three-dimensional world. Once it has learned to do this, such movements will seem to become automatic, requiring no conscious thought, and we would then cease to think of them as needing intelligence.

The robot arm is in the same position as the human toddler — it has the equipment to perform tasks, but it must 'learn' how to perform these tasks automatically.

The simplest method is to train the arm to perform specific tasks merely by leading it through a sequence of movements and, in effect, telling it to 'remember this'. This method is used with a large number of industrial robots. An operator literally takes the robot by the hand and leads it through the steps that it must follow. This has the big advantage that the person needs to know nothing about how the robot arm actually works — all he must know is the sequence of actions that the arm must follow. In turn, the robot does not need to 'know' what it is doing — it simply has to 'remember' the actions it must carry out.

TRAINING METHODS

There are two types of 'training' used with robot arms — point-to-point training and continuous path training. In point-to-point training, the operator moves the arm to a certain position and then presses a button to signal to the robot that this position must be 'remembered'. The arm is then moved to the next position and the button pressed again. This sequence is continued until a whole sequence of actions has been stored in the robot's memory. Once the training session is over, the robot may be switched into 'playback' mode, and it will then move from point to point in exactly the way it was 'taught'. In continuous path training, the operator simply leads the robot through the complete sequence, and the robot remembers each and every position in the sequence. On playback, the robot will follow the sequence in the same way as before.

Returning to the example of a person sitting at a table and moving an object from one side of it to another, we can use the point-to-point method to 'train' a robot to duplicate this action. This method is often used with robots that must carry out 'pick and place' tasks — moving an object from one location to another. By contrast, a robot that is used for spray painting will need to be taught by the continuous path method to ensure that it covers the whole object evenly with paint, just as a person would do.

We now need to consider how the robot 'remembers' the sequence of movements that it should follow. The answer is that the robot uses its internal sensors to record the position of each of its joints during the training mode. This is often done by taking the output from the shaft encoders in the robot joints and recording the movements made, either directly into memory or, for more permanent storage, on tape or disk. When playback mode is selected, the robot can then recall all the relevant data and convert it into joint movement — a relatively complex task.

Surprisingly, it is easier for the robot to 'remember' continuous path movement — it has only to follow the exact route it has been taught. However, a very large amount of data needs to be stored — often several thousand positions are needed to define a continuous path, rather than the very few positions that are involved in point-to-point movement. The second difficulty arises from the fact that, if the arm is to follow the sequence smoothly and exactly, all of its joints must be activated simultaneously. A spray-painting robot may have to make a sweep of the arm along all three axes, while manoeuvring its three wrist joints to position the spray correctly. This means that the computer controlling the robot must work very fast in order to manipulate each joint in turn with no appreciable delay; alternatively, the robot may use as many as six separate processors, each directing the movement of one joint, to achieve truly simultaneous movement.

CALCULATED MOVE

A point-to-point robot has a harder task because, although it knows where it should move to, it has not been 'told' how it should get there. It could simply move each joint until it was at the required position, but this would be wasteful of both time and energy. It would be much better if the robot could calculate a direct route for its hand from one point to another; it could then make the required movement in one sweep, just as a person does. But the calculations required to do this are again complex, as Cartesian co-ordinates must be used to move the hand in a straight line between two defined points, while the arm positions themselves are defined in a different co-ordinate system

Two-Joint Geometry

In moving from point to point, the two-joint robot arm must rotate about its pivot (angle R), and it must change the shoulder (S) and elbow (E) angles. If the Cartesian co-ordinates of the current and destination points are (X1,Y1,Z1) and (X2,Y2,Z2) then the changes are calculated as follows:

```
A1=SQR(X1^2+Y1^2+Z1^2)
A2=SQR(X2^2+Y2^2+Z2^2)
```

Pivot:

```
R1=ARCTAN(Y1/X1)
R2=ARCTAN(Y2/X2)
Change=(R2-R1)
```

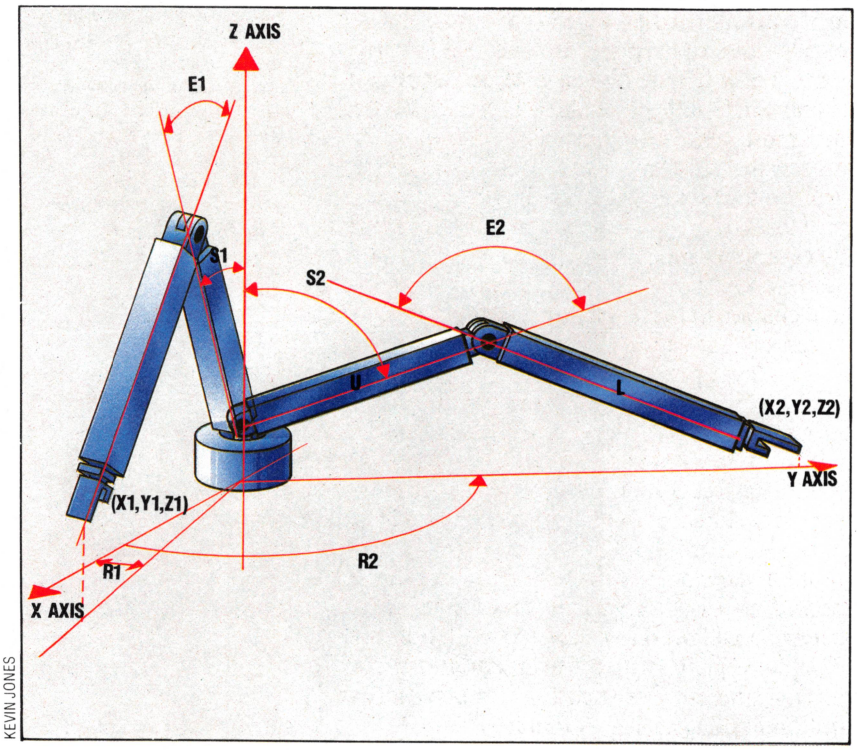
Shoulder:

```
S1=ARCCOS(Z1/A1)+ARCCOS
      ((A1^2+U^2-L^2)/
      (2*A1*U))

S2=ARCCOS(Z2/A2)+ARCCOS
      ((A2^2+U^2-L^2)/
      (2*A2*U))
Change=(S2-S1)
```

Elbow:

```
E1=ARCCOS((U^2+L^2-
A1^2)/(2*U*L))
E2=ARCCOS((U^2+L^2-A2^2)/
      (2*U*L))
Change=(E1-E2)
```

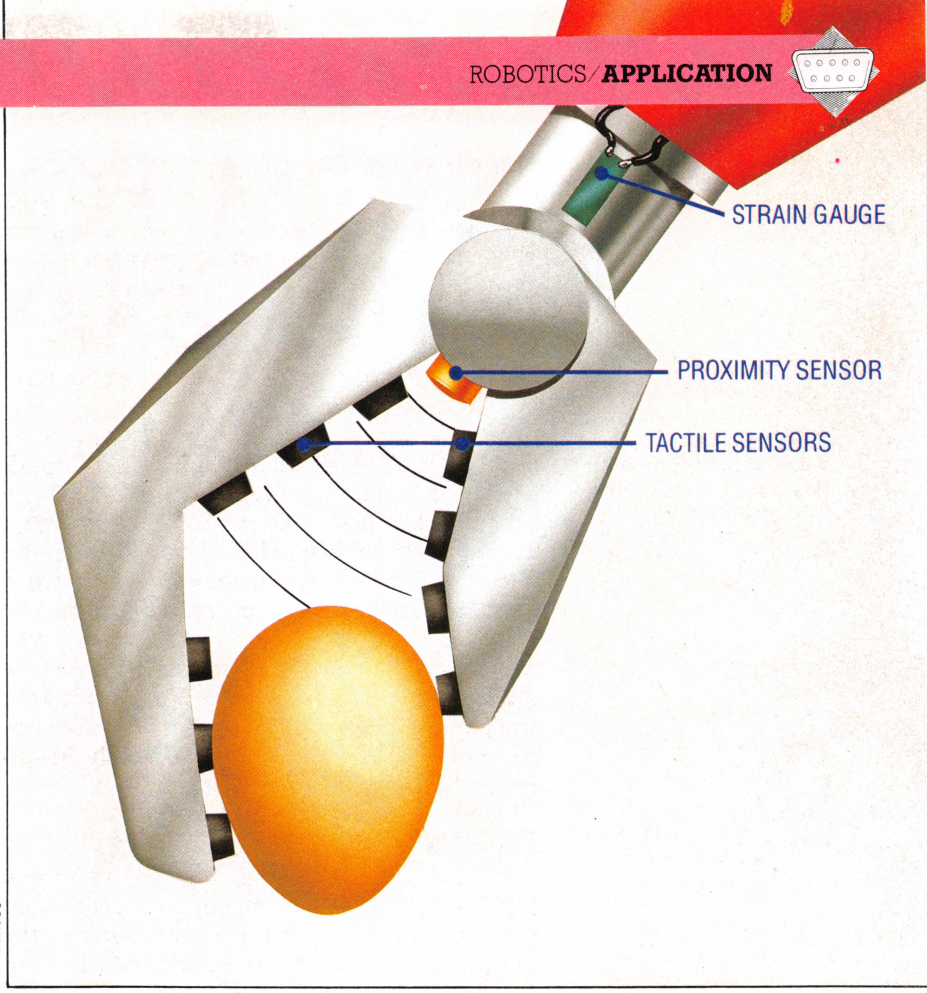
where U and L are the lengths of the upper- and lower-arm respectively



KEVIN JONES

entirely — typically revolute co-ordinates. So the robot must be able to solve some tricky geometrical problems in order to work efficiently. And, in the case of industrial robots that must move objects weighing several hundred kilograms over a large distance, the saving in time and energy by choosing the best route can be considerable.

Another problem facing the point-to-point robot is the dynamics of the arm itself. Move your own arm to pick up an object and you will find that it accelerates slowly away from its original position until it reaches maximum speed, then decelerates until it comes to a smooth stop at its final position. The advantages of a robot arm that does this are considerable. Many such arms move at a constant speed, accelerating almost immediately to maximum velocity and stopping dead at the end of the movement sequence. This places strain on the arm itself and requires more power than an arm which accelerates and decelerates smoothly. It also means that the arm may not move as quickly as it could otherwise do and, if at the end of the sequence the robot were required to pick up a delicate object, even a slight displacement of the object could result in the arm hitting it with considerable force. So the robot needs to work out an optimum speed as well as an ideal path to follow.



STEVE CROSS

Gently Does It
Picking up an egg is a searching test of the robot arm's sensors and feedback control mechanisms. The gripper's proximity sensor must check that the egg is close enough to grasp, then the fingers can start to close until the touch sensors indicate contact with the egg. The output of the touch sensors must now be checked against that of the proximity sensor as the fingers close and the arm begins to lift. A sudden decrease in proximity shows that the egg is slipping, so the fingers must tighten until a preset grip-force limit is reached or until sudden decrease in grip-force shows that the eggshell is distorting prior to cracking

ROBOT CHOREOGRAPHY

Even after the arm has been programmed to follow a precise set of movements, it may turn out on playback that these movements were not quite what was required. This may be because of human error, because the nature of the task has changed slightly, or because, if the robot is working in conjunction with other arms, the arms may follow their own paths and collide with each other. (Avoiding this latter problem is known as 'robot choreography'.) So a method of editing the sequence is required. This can be achieved by storing the movements as a linked list, in which each position is stored and followed by the address at which the next postion is to be found. In the initial training session, this address will be the address of the next position in the list. If the sequence needs to be edited, the arm could be moved to the position at which corrections need to be made, stopped, and a new sequence inserted.

Another common method of making arms move intelligently is to use a series of programmed instructions stored in the computer. Typically, each robot has its own programming method and uses a different programming 'language' to control movement, but in general a language is required that enables the programmer to use LOGO-like commands to specify movement in three dimensions, with added instructions for wrist and end effector movement, such as 'pick up' or 'put down'.

The problem is similar to training a point-to-point robot, and many factors need to be taken into consideration. For instance, if a robot arm is to move forward 10 units then the obvious method would be to alter the shoulder joint so that the arm can reach further forward. However, this would cause the arm to move upwards in an arc, and so this must be corrected by a downward movement in the elbow joint. From this it can be seen that the instruction for just one simple movement must be translated into two distinct sets of instructions, working on two separate joints.

Other problems can arise when the robot is required to pick up an object. However well the arm is positioned, it is difficult to ensure that it is in exactly the right place to pick up the object, particularly if that object is of an asymmetrical shape. An 'intelligent' hand is therefore needed — this must sense the presence or absence of the object, the distance of the object from the hand, and the force exerted by the hand when it tries to pick the object up. These problems may be tackled by equipping the hand with a range of proximity, tactile and force sensors, which provide feedback that enables the controlling computer to make any necessary corrections.

If all these problems are considered, we can see that it is possible to construct a robot arm that shows a relatively high degree of 'intelligence'. However, as yet no arm can be designed to, say, bowl a cricket ball accurately. This is because the arm's intelligence alone is not enough. The robot must also know the state of the pitch, the position of the batsman, the wind strength and direction, and a host of other variable conditions. Then, of course, it will need to be able to work out the very complex equations involved in sending a projectile through the air. For such tasks, much more than just an intelligent arm is required.

ROOM FOR MANOEUVRE

Simple utility programs, like the variable search program we wrote on pages 664 and 700, can be written entirely in BASIC, using only information about how individual lines of BASIC are stored. For more complicated utilities, however, we need greater detail and therefore must resort to machine code.

In order to operate our variable search program, we merged it with the program to be searched. With this method, the only information from the operating system that we had to supply was the address where the BASIC program starts; the end of the program being searched was found by testing for the lowest line number in the utility program.

The utility that we are creating is a variable replace program. This is a very useful program to have on file. If you had used a variable name throughout a program only to find it was illegal, imagine how much time such a program would save. Similarly, you might have written a program you wanted someone else to use in which the variable names were not easy to decipher. Here we explain the necessary theory for the machine code and in the next instalment we will publish the listings.

SPACE IN MEMORY

In this exercise, we need to put the utility program in a separate section of memory from the program it is working on. We must also find a different method of locating the end of the BASIC program, and a means of accommodating two BASIC programs in the computer at the same time.

The three computers that we are looking at — the BBC Micro, Commodore 64 and Sinclair Spectrum — use a set of pointers to tell the operating system and the BASIC interpreter where to locate BASIC programs and variables, etc. (see page 56). Unfortunately, the details are different in the three machines.

On the BBC Micro, there are four important pointers: PAGE and TOP, which hold the beginning and end address of the BASIC program; LOMEM, which holds the start address of the BASIC variables; and HIMEM, which holds the end address of the BASIC area. These four pointers are stored as built-in BASIC variables, and we can read or alter their values by simple BASIC statements. If we have a BASIC program in memory and we wish to add another, we change PAGE to a value higher than TOP — using the command OLD to reset TOP and LOMEM — and can then add the new program without affecting the original program. We change from one program to another by giving new values to both PAGE and HIMEM and using the command OLD.

Once we have the utility program running, the values of the pointers refer to the utility program; to enable the utility to find the start and end of the program it is to work on, we need to copy the original values into an area of memory that will not be altered when we change programs. Another method of finding the end of a program is to use the end marker that the BASIC interpreter puts in. This is simply a byte holding a value of 128 or more, immediately following the carriage return character at the end of the last line of the program. This byte, and the one following, will be interpreted as the HI and LO bytes of the next line number. Since the HI byte of this number is 128 or more, this will give a line number of 32768 (256×128) or more. As the highest valid line number is 32767, we can be sure that we have found the end of program marker and not just another line number.

The Commodore 64 uses seven pointers, stored in zero page memory, to indicate various parts of the BASIC program area. TXTTAB, at addresses 43 and 44, points to the start of the BASIC program; VARTAB, ARYTAB, STREND, FRETOP and FRESPC, at addresses 45 to 54, point to various sections of the variable table; and MEMSIZ, at addresses 55 and 56, points to the end of the BASIC area. It is possible to change these pointers in order to create a separate area in which to run a BASIC program by use of the POKE command. However, a short machine code program is recommended as it is more direct, and reduces the chances of crashing the computer with a typing mistake.

In the Commodore 64, the end of a BASIC program is indicated by two bytes containing zeros immediately following the zero byte marking the end of the last line of the program. Following the chain of pointers at the beginning of each line of the program until we find a pointer of zero will indicate the end of the program.

FOR THE SPECTRUM

Creating this utility is rather more complicated on the Spectrum. Instead of a separate area for the BASIC program, there is a single, continuous block of memory that includes not only the BASIC program and variables, but also all the workspace areas used by the operating system and the BASIC interpreter. With this layout of the memory it is difficult, if not impossible, to have two BASIC programs in the main working area, so we will make a copy of our program above RAMTOP and work on it there. This still leaves the problem of recovering the program and fitting it into the main

program area after it has been altered, and we will need a machine code program to do this for us.

The Spectrum manual gives a great deal of information about the way a BASIC program is stored and what the various areas of memory are used for. However, because of the large number of different sections in the working area, and the way these areas can move around, it is difficult to write utility programs without using machine code subroutines from the ROM. If you want to do any serious utility programming on the Spectrum, a valuable reference work is *The Complete Spectrum ROM Disassembly*, by Dr Ian Logan and Dr Frank O'Hara. This explains how all the ROM routines work.

Two of the most important subroutines in the ROM for use in utility programs are the routines that open up or reclaim space in the working area, and we will be looking at these when we come to the variable search and replace program.

Experimenting With BASIC

You can try altering the contents of a program during execution but you should save the program first, as a system crash is a common result. Use the Monitor program (see page 118), which allows you to inspect and alter the contents of memory. This can inspect and alter itself under your command. Insert some extra REM lines at the start of the program and try these suggestions on them first:

- Find the start of BASIC text area (see page 58), and inspect the Monitor program in memory until you can identify program lines.
- Change the values of the bytes after a REM token, then quit the program and list the altered line.
- Try putting a value greater than 127 into a REM line — again, quit and list: you may be surprised.
- Alter the line number bytes of a line — this produces unpredictable results, especially if the new number is out of sequence with its neighbours.
- You can alter the line length bytes — try decreasing the indicated length first — but you should insert a new end-of-line marker at the indicated byte.
- On the Commodore 64 you can change the link address bytes: try replacing one line's link address by the link address of the succeeding line, and then list the program.
- If you're feeling more ambitious, consult your manual and explore the variables' storage area. This usually begins in the memory map where the BASIC text area ends. There are up to six different variable types, each with its own storage format: numeric variables, numeric arrays, integer variables, integer arrays, string variables and string arrays. String and integer variable formats are the simplest, being essentially straightforward representations of the data and the variable name; numeric array data is the most complicated.
- You can try changing token values in program lines: this will change the command word. If you do this via the Monitor program to a line that it is currently executing, you will be introducing a potentially massive paradox into the interpreter

ERRATA

On page 118 in the BBC and Commodore BASIC flavours:

- In line 1150 there are two successive assignments to C$(3) — change the second assignment to
  C$(4)="Q"
- In line 6600 change Z=1 to Z=2
- Line 200 of the BBC flavours should be
  200 CLS

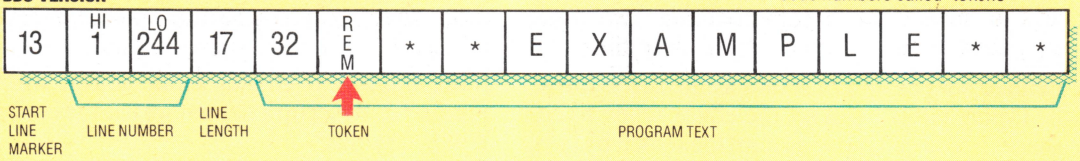
as in the Spectrum version

How BASIC Programs Are Stored

Most micros follow essentially the same storage format in the BASIC program area. Each program line begins with the line data — BASIC line number in two-byte form, and some information about the length of the line. Program text is stored more or less unaltered, though BASIC keywords are replaced by one-byte code numbers called 'tokens'

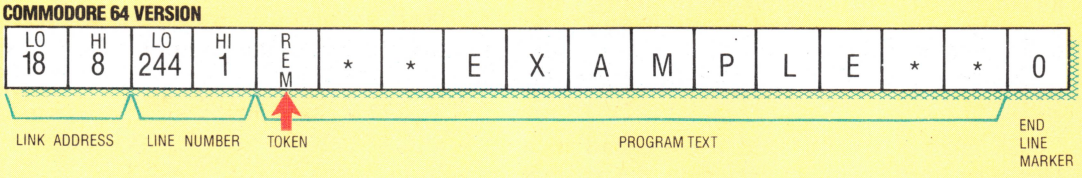
500 REM * * EXAMPLE * *

BBC VERSION



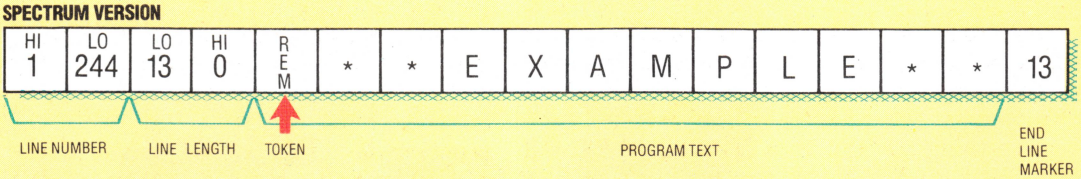
In this format 13 (ASCII for [RETURN]) is a start-of-line marker; it is more commonly placed at the end of the line. Line length has only a single byte, which confines program lines to 255 characters. The space directly after the line number has been stored

COMMODORE 64 VERSION



The Link Address bytes contain the two-byte address of the first byte of the next program line. The BASIC program text area starts at address 2049, and this line is 17 bytes long, so the start address of the next line is 2066

SPECTRUM VERSION



Spectrum methods are always interesting: the line length occupies two bytes, so a single program line could be 65,535 characters long! The line length here is 13 — the bytes in the line including the end-of-line byte, but not counting the line data bytes

LIZ DIXON

WOOLLY JUMPERS

In this instalment we look at the sprite handling facilities of Atari LOGO. These are exceptionally good, and include commands to define your own sprites and determine their speed. There are a multitude of colours to choose from, and a novel facility for detecting events — like collisions — and taking evasive action.

Atari LOGO has four sprites, which are numbered 0 to 3, with 0 as the 'default' turtle. The Atari manual refers to them as turtles rather than sprites, so we'll call them turtles too.

TELL 1 makes turtle 1 *current*; in other words, turtle 1 will obey any commands you give. Try:

```
TELL 1
FD 40
RT 90
TELL 2
BK 40
RT 90
TELL 3
RT 90
```

You can, however, have more than one current turtle. Try:

```
TELL [1 2 3]
FD 50
```

Now those three turtles will obey the commands.

All four turtles have the rotating turtle shape, shape 0, until another shape is defined. Up to 15 other shapes can be defined, using the editor, and then assigned to the turtles. These user-defined shapes do not rotate as the turtle turns.

Typing EDSH 1 will set the editor ready to edit shape 1. You can move around the screen by using the cursor keys. Pressing the Space bar will switch an empty box to full or a full box to empty. Once you have designed a shape, you define it by pressing <ESC>. The command SETSH 1 will give turtles 1, 2 and 3 (the current turtles) the new shape.

ASK enables you to send a command to a particular turtle without changing the current turtles. Try:

```
ASK 1[FD 20]
```

and you will see that only turtle 1 moves. Now type FD 20 and all three turtles will move because the current turtles are still numbers 1, 2 and 3.

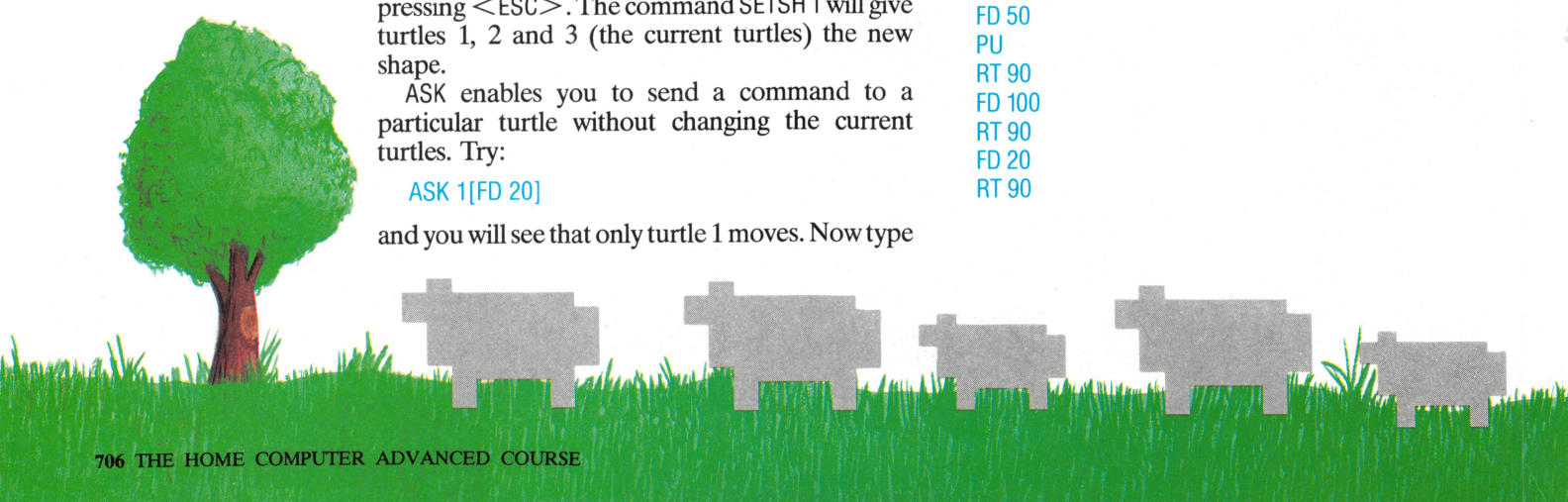
Turtles can be given a speed, as well as the heading and a position. SETSP 30 gives the current turtle a speed of 30 in its present direction. The turtles will keep their speeds until you change them. You can run procedures, or create drawings on the screen, and the speeds are unaffected. To stop the turtles, you give them a speed of 0. Entering the editor will stop them as well, since entering the editor will always destroy the graphics display because they share the same area of memory.

On the Atari you have 128 different colour shades to choose from. You can set the colour of the background, the pen colour and the colour of the turtle. For example, SETBG 92 will set the background green. SETPC 0 23 will set the pen colour to orange. The 23 is the code for orange, and 0 is the pen number. In Atari LOGO, the turtle has a choice of three pens with which to write, although we shall use only pen 0 (the default pen) in this article. SETC 7 will set the current turtle white. There is a table of colours and corresponding colour codes on page 26 of the Atari reference manual.

DEMONS

The most original aspect of Atari LOGO is its use of *demons*. There are 21 *collisions* and *special events* (see page 145 of the Atari reference manual) that LOGO can detect. Most of these are collisions between turtles, or turtles and lines. A demon tells LOGO what to do when one of these collisions occurs. For example, collision number 0 is when turtle number 0 crosses a line drawn with pen number 0. To set up a demon we use the command WHEN. Try this:

```
CS
TELL 0
PD
FD 50
PU
RT 90
FD 100
RT 90
FD 20
RT 90
```



with the turtle facing it. Now set up a WHEN demon with the following command:

```
WHEN 0 [BK 50]
```

Nothing happens immediately, but the WHEN demon is now present inside the computer keeping a watch out for event 0. Now try SETSP 30. The turtle sets off towards the line, but when it reaches it the WHEN demon is triggered and the turtle is thrown back. The turtle keeps on going with a speed of 30, but every time it comes to the line the WHEN demon repulses it.

This WHEN demon will stay in operation until you clear it by typing WHEN 0 []. All demons will be removed if you type CS, if there is an error message, or if you use the editor.

It is a nuisance to have to remember all the codes for the various collisions, so there are two primitives to help you: OVER <turtlenumber> <pennumber> outputs the number for the collision between that turtle and a line drawn in that pen, and TOUCHING <turtlenumber 1> <turtlenumber 2> outputs the number of the demon for a collision between those two turtles.

CAGING TURTLE

Here is a set of procedures to cage a turtle within a box. Whenever the turtle hits the edge of the box (WHEN OVER 0 0) a demon calls up the procedure TURN. This causes the turtle to retreat 10 units, and then make a random turn. (RANDOM, together with a number, N, outputs a random number between 0 and N−1 inclusive).

```
TO TRAP
   DRAW. TRAP
   HOME
   WHEN OVER 0 0 [TURN]
   SETSP 50
END

TO DRAW. TRAP
   CS
   PU
   SETPOS [-50 -50]
   PD
   SQUARE
   PU
END

TO SQUARE
   REPEAT 4 [FD 100 RT 90]
END

TO TURN
   BK 10
   RT RANDOM 45
END
```

Demons can also be used for watching the joystick. Of the 21 special events that we have mentioned, event 3 occurs when the joystick button is pressed, and event 15 occurs when the joystick position is changed. The command JOY 1 outputs a number from -1 to 7 corresponding to the position of the joystick (in port 2). Define JOYH in this way:

```
TO JOYH
   IF ( JOY 1 ) < 0 [STOP]
   ASK 0 [SETH 45 * JOY 1]
END
```

then set the turtle in motion with SETSP 50, and finally set up a WHEN demon:

```
WHEN 15 [JOYH]
```

The joysticks can now be used to control the heading of turtle 0.

You can give more than one WHEN command at the same time, but they are not actually active simultaneously. While one demon is busy (i.e. when its event occurs) the others are inactive. This can mean that some collisions go undetected.

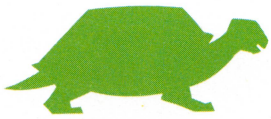
The way to deal with this problem is to have each demon simply set the speeds to zero and to run a continuous procedure that watches for this happening. Adapting our previous program to use this technique:

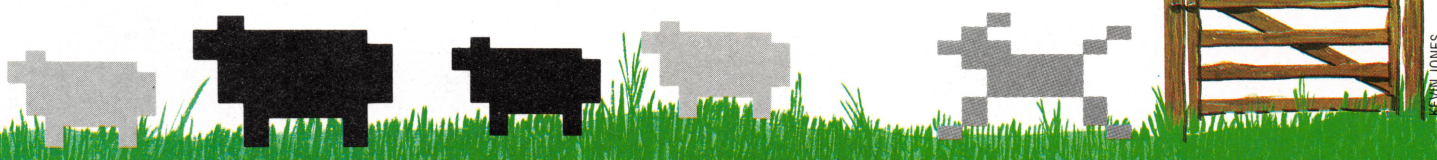
```
TO TRAP
   DRAW. TRAP
   HOME
   WHEN OVER 0 0 [SETSP 0]
   SETSP 50
   WATCH
END

TO WATCH
   IF :SPEED = 0 [CHECK]
   WATCH
END
```

In this procedure SPEED outputs the value of the speed of the current turtle. The procedure CHECK must determine which event has occurred, carry out the necessary actions and then restore the speeds. In this case there is only one event we are interested in, but it illustrates the way to program the method.

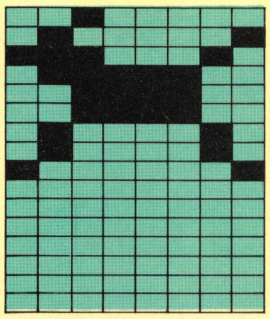
```
TO CHECK
   IF COND OVER 0 0 THEN [TURN]
   SETSP 50
END
```

The command COND and a number gives a true output if an event of that number has occurred. COND can only check an event at the time LOGO executes the line containing it.

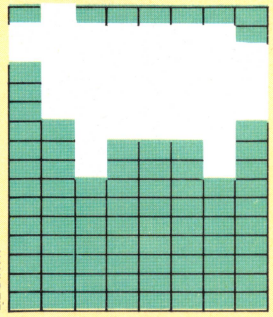




KEVIN JONES



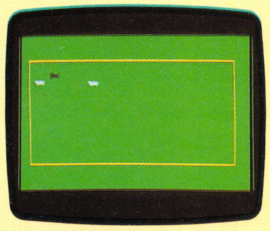
Canis Familiaris — The Dog



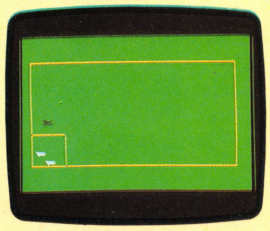
LIZ DIXON

Ovis Aries — The Sheep

Game Start



The Middle Game



IAN McKINNELL

Back In The Pen Again!

ROUNDING UP SHEEP

We give a game that uses many of the features we've described. The player uses the joystick to control a dog that is chasing two sheep in a field. If the sheep run into the fence they will back off and turn. If the sheep run into one another they will turn at random. If the dog touches the sheep they turn 90° to the right. Pressing the joystick button causes a little cage to be drawn in the bottom left-hand corner of the field. Pressing the button again will erase the cage. The dog's task is to manoeuvre the sheep into the cage.

```
TO CHASE
   SET.VAR
   ASK :TURTLE [SET.SCREEN]
   SET.DEMONS
START
   WATCH
END

TO SET.VAR
   MAKE "FENCE 0
   MAKE "TURTLE 0
   MAKE "SHEEP1 3
   MAKE "SHEEP2 2
   MAKE "DOG 1
   MAKE "GREEN 92
   MAKE "ORANGE 23
   MAKE "BLACK 0
   MAKE "WHITE 7
END

TO SET. SCREEN
   CS
   FS
   SETBG :GREEN
   HT
   PU
   SETPOS [-150 -80]
   PD
   SETPC 0 :BROWN
   RECT 160 300
   PU
END

TO RECT :SIDE1 :SIDE2
   REPEAT 2 [FD :SIDE1 RT 90 FD :SIDE2 RT 90]
END

TO SET.DEMONS
   WHEN OVER :SHEEP1 :FENCE [SETSP 0]
   WHEN OVER :SHEEP2 :FENCE [SETSP 0]
   WHEN TOUCHING :SHEEP1 :SHEEP2 [SETSP 0]
   WHEN TOUCHING :DOG :SHEEP1 [SETSP 0]
   WHEN TOUCHING :DOG :SHEEP2 [SETSP 0]
   WHEN 3 [SETSP 0]
   WHEN 15 [JOYH]
END

TO JOYH
   IF ( JOY 1 ) < 0 [STOP]
   ASK :DOG [SETH 45 * JOY 1]
END

TO START
   SET :SHEEP1 1 [-150 20] 45 :WHITE
   SET :SHEEP2 1 [150 20] 315 :WHITE
   SET :DOG 2 [0 0] 0 :BLACK
   SET.SPEEDS
END

TO SET :NO :SHAPE :POS :HEAD :COLOR
   TELL :NO
   PU
   SETSH :SHAPE
   SETC :COLOR
   ST
   SETPOS :POS
   SETH :HEAD
END

TO SET.SPEEDS
   ASK :SHEEP1 [SETSP 10]
   ASK :SHEEP2 [SETSP 10]
   ASK :DOG [SETSP 60]
END

TO WATCH
   IF SPEED = 0 [CHECK]
   WATCH
END

TO CHECK
   IF COND OVER :SHEEP1 :FENCE [ASK :SHEEP1
   [BK 20 RT 90]]
   IF COND OVER :SHEEP2 :FENCE [ASK :SHEEP2
   [BK 20 RT 90]]
   IF COND TOUCHING :SHEEP1 :SHEEP2 [BUMP]
   IF COND TOUCHING :DOG :SHEEP1 [ASK
   :SHEEP1 [RT 90]]
   IF COND TOUCHING :DOG :SHEEP2 [ASK
   :SHEEP2 [RT 90]]
   IF COND 3 [ASK :TURTLE [DRAW.CAGE]]
   SET.SPEEDS
END

TO BUMP
   ASK :SHEEP1 [SETH RANDOM 360]
   ASK :SHEEP2 [SETH RANDOM 360]
END

TO DRAW.CAGE
   PU
   SETPOS [-150 -30]
   PX
   SETH 90
   REPEAT 2 [FD 50 RT 90]
   PU
END
```

Logo Exercises

1. Change the sheep herding game so that the dog is controlled by using the keyboard rather than the joystick.
2. Write a program for a game in which you are in control of a spaceship. Meteorites come hurtling towards you and you must dodge out of their way and survive as long as possible. Here are some hints to help you. Use one sprite for the ship and the others for the meteorites. Use WHEN demons to check collisions. The meteorites move at a steady speed but in a random direction. The spaceship can be controlled by the joystick.

BETTER BY FOUR

The Commodore 64 is one of the world's top selling computers, so it might seem that the company would be hard put to improve on it. However, Commodore's latest home computer, the Plus/4, represents an improvement in several ways. It has a better version of BASIC, four built-in applications programs, and a full 64 Kbytes of memory.

Commodore claims that the Commodore Plus/4 is intended to sell alongside the Commodore 64 and not to replace it. But the new model offers so many improvements on the 64 that if it succeeds in the marketplace, it could well supersede its predecessor entirely.

The Plus/4 uses the 7501 microprocessor, which is a development of the 6502. This chip is designed in such a way that it can access more than 64 Kbytes of memory. This means that the machine has room for a decent BASIC, while keeping its RAM free for the user. There are 64 Kbytes free to be used by BASIC programs, although this falls to 50 Kbytes when graphics are used. This is better than every other home micro, except for the Sinclair QL (see page 501), and the Advance 86a (see page 349).

An excellent version of Microsoft BASIC has been implemented on the Plus/4, and the graphics and sound commands are particularly worth noting. In graphics mode, the DRAW command will produce dots or lines and any outline shape can be filled with colour by the PAINT command. The BOX command will draw squares and rectangles in outline or in solid colour. The CIRCLE command is particularly versatile. As well as drawing circles, ovals can be created by specifying the height and width of the oval. Parts of ovals may be drawn to produce arcs, simply by specifying start and stop positions in the command.

All commands normally operate on a screen with a resolution of 320 by 200 dots. This is the same resolution as the Commodore 64, but the Plus/4 really excels in its choice of colours. It can show 120 different colours, plus black, on the screen at the same time. These are created from 15 basic shades, each of which can be displayed in eight different brightnesses. Unfortunately, the Plus/4 cannot produce sprites.

The commands to control sound are fairly standard. The SOUND command plays a note of specified pitch and duration. A separate command, VOL, specifies one of eight settings for the volume of each sound channel. All sound is output through the television loudspeaker. Only two sound channels are provided on the Plus/4, although the BASIC allows three channels to be specified. This 'third' channel is, in fact, a noise facility and any note given that channel reference will be reproduced as noise. This is very useful for games, when special sound effects are required.

CHRIS STEVENS

Meet The Future?
Commodore's long-awaited successor to the Commodore 64 has all the features that are becoming standard in the latest micros: MSX-style looks and cursor cluster, big memory and on-board software. The sales competition, however, is intense, and potential customers are more informed than ever. Commodore is not taking its market dominance for granted, as the Plus/4's looks and features plainly show

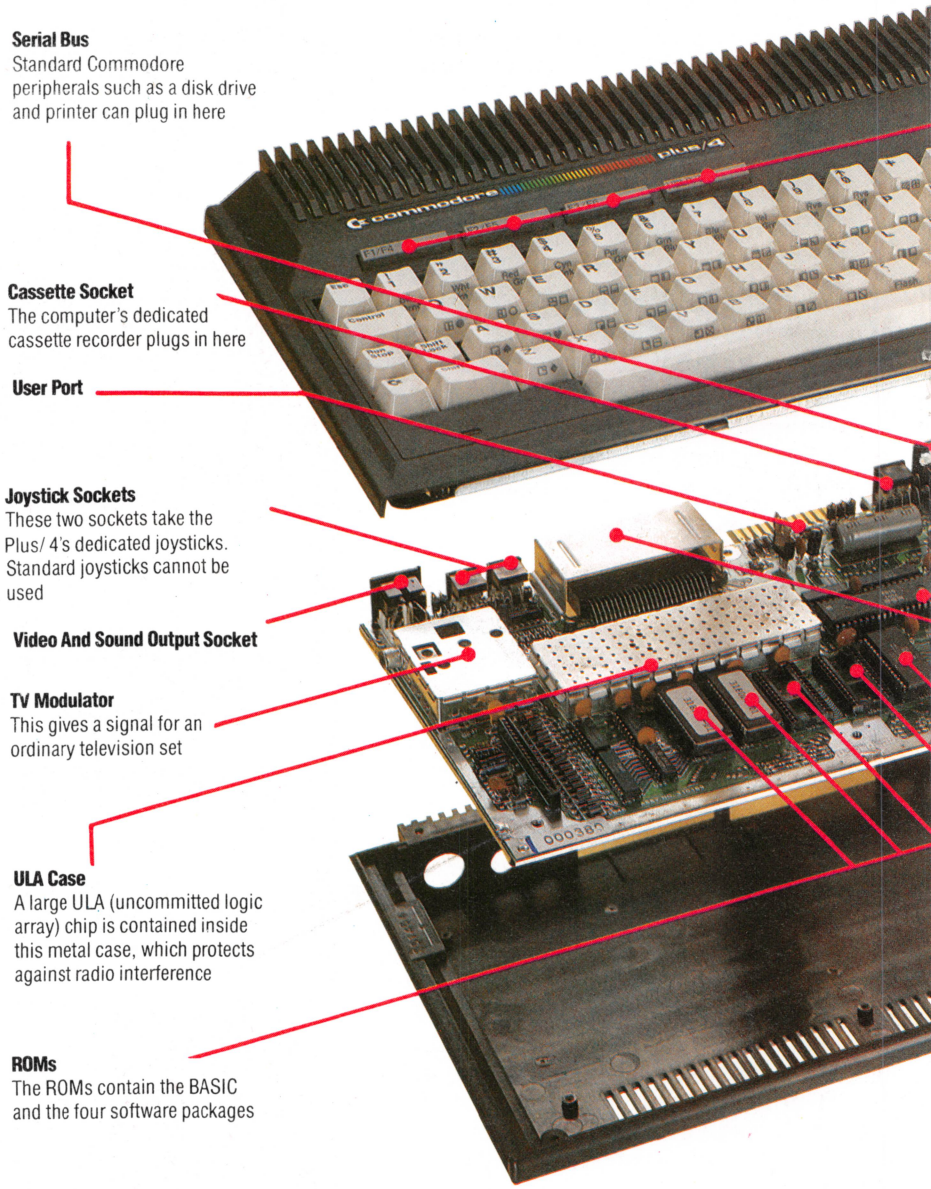
A number of commands have been included to improve the main body of the BASIC language. An AUTO command will produce line numbers automatically when programs are keyed in; RENUMBER will give new line numbers to programs; and VERIFY will check that programs have been successfully saved on cassette or disk. There are many new commands for working with disks, and Commodore evidently hopes to sell disk drives to a high percentage of Plus/4 owners.

The Plus/4 has a text display of 40 by 25 characters. The user can specify two points on the screen to act as the corners of a 'window'. All text, such as listings and commands, will then appear within that window area only, leaving the rest of the screen untouched.

The keys on the Plus/4 are very sensitive to touch, needing only the slightest pressure for them to register. A number of characters such as @, =, +, −, and £ are given their own keys and a full set of graphics characters can be produced from the keyboard. At the top of the keyboard are four function keys, and when the machine is turned on, they are automatically set up to produce the most commonly-used commands. The function keys

QL Vs Plus/4
Objectively, there is no comparison: the QL with its built-in Microdrives, bigger memory, SuperBASIC and superb software is, by market standards, a snip at £400; the Plus/4 with cassette recorder for about £350 scores only on the quality of its keyboard. A cool appraisal would indicate the QL without hesitation, and yet most of us will finally decide entirely on 'feel' and brand loyalty. Without doubt the Plus/4 will sell and sell!

IAN McKINNELL



enable the user to define a new command of up to 128 characters for each key. Eight different functions can be produced from the four keys by using them with the Shift key.

The generous amount of memory space allows the Plus/4 to have built-in applications software. Four programs are provided: a word processor, spreadsheet, database and graphics package. These programs are designed to work together.

Unfortunately, the word processor is rather difficult to use. The Plus/4 can display only 40 characters across its screen width, but many printers can produce 80 characters. To match this width, the screen pans sideways once the 37th column is reached and continues panning until the 77th column, when it jumps back to the first column. The program has formatting commands to set margins and justify text, but these come into effect only when the text is printed out. It also has SEARCH and REPLACE commands to locate particular words or phrases within a document and replace them if necessary. A limit of 99 lines is set on the amount of text that may be entered. The 77-character line width means that this is a maximum of 1,500 words, which is not sufficient for serious applications.

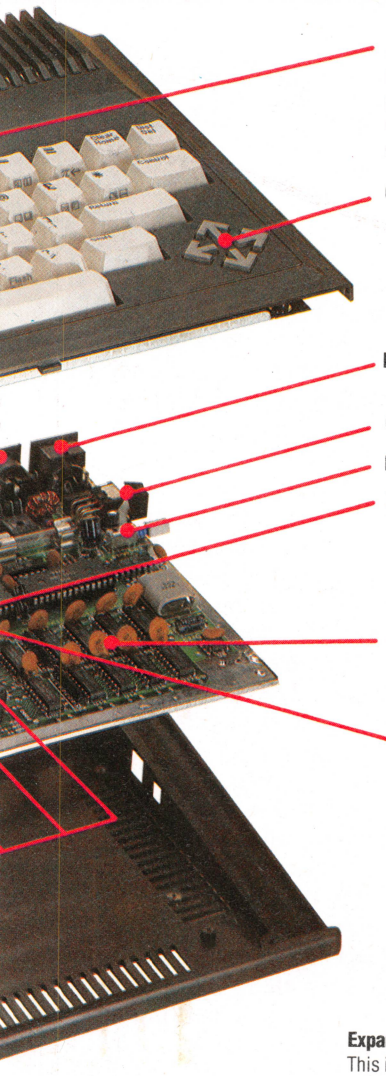
The spreadsheet program is easier to use than the word processor, although it also suffers from the limitations of a 40-column screen display. This means that it can show only three spreadsheet cells across the width of the screen and 12 down, even though it can handle models up to 17 cells wide and 50 deep.

The graphics program is rather disappointing. All it does is transform a set of figures from the spreadsheet into a kind of crude bar graph, made up from block graphics, and transfer it to the word processor. There it can be displayed or printed.

Both the word processor and spreadsheet can be used with the standard machine, but the only way to save their results is with a disk drive. This means they are of little use to people who rely on cassette storage. The last program, the database, cannot be used at all without a disk drive. It works by defining a standard format that is recorded onto disk as empty records. All data is then entered into the empty records. This means that one disk can be used for one database only and the format of the data cannot be changed once data has been recorded. Each database can hold up to 999 records, each with up to 17 fields of 38 characters. Generally, the software is disappointing, being too crude for serious business use, and requiring the home user to buy a disk drive.

Many home users will be far more pleased with the built-in machine code monitor, Tedmon, than with the software. The monitor is a great help to the machine code programmer and is called into use by the command MONITOR.

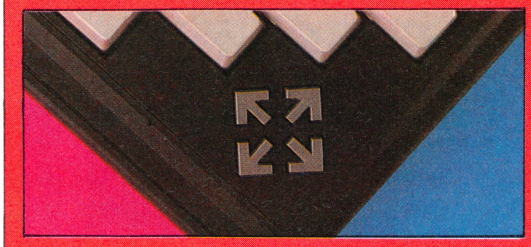
Commodore is producing a number of add-ons for the Plus/4. The most important of these for many users will be the cassette recorder. Like other Commodore micros, the Plus/4 needs a cassette recorder made specially by Commodore.



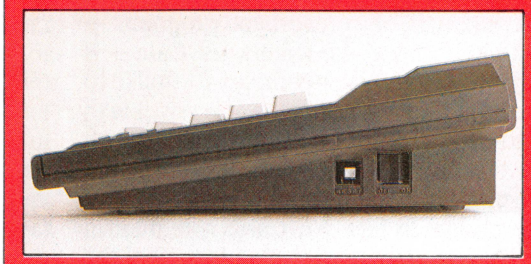
CHRIS STEVENS

Function Keys

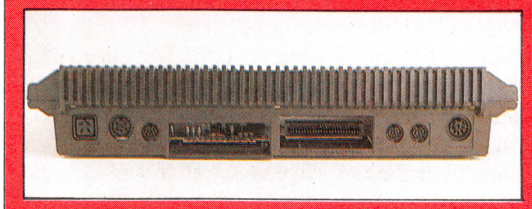
Cursor Cluster



Reset Button



Expansion Port / User Port



New Features

Old Commodore hands will be interested to learn that the Plus/4 has both Escape and Reset keys. Other encouraging signs are the cursor keys cluster, the function keys (programmable from BASIC), and the Help key. The power supply, cassette and joystick sockets are all different from the Commodore 64 and Vic-20 standard, as are the expansion and user ports

COMMODORE PLUS/4

PRICE
£300

DIMENSIONS
67 × 203× 338 mm

CPU
7501, 0.9 or 1.8 MHz

MEMORY
64K RAM, 64K ROM

SCREEN
Text: 40 by 25. Graphics: 320 by 200 in 121 colours

INTERFACES
2 joystick sockets, serial interface, cassette interface, cartridge/ parallel port

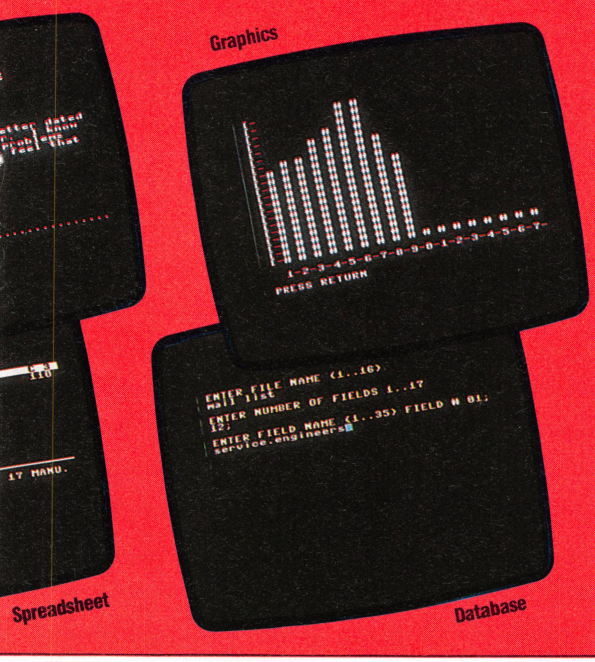
LANGUAGES AVAILABLE
BASIC

KEYBOARD
Typewriter-style with 67 keys, including four function keys

DOCUMENTATION
Reasonable manuals that teach how to program and how to use the built-in software. Badly phrased in places

STRENGTHS
Particularly good version of BASIC with good graphics and structuring commands. Full 64K of memory

WEAKNESSES
Needs dedicated cassette recorder and non-standard joysticks. Built-in software is poor and cannot be used fully without a disk drive



IAN McKINNELL

Hidden Cost

The Plus/4's ROM-based software comprises a word processor, spreadsheet, database and graphics utility. The packages' chief strengths are the fact that they are integrated; they are not, however, usable without a disk drive, which effectively increases the cost of the machine by £200

It uses a different cassette plug to the older Commodores so a different recorder has to be bought. The price is the same, though — £45. The Plus/4 also uses a different socket for joysticks.

A slightly updated version of the slow Commodore disk drive is being offered for the machine. Commodore is also developing a 'fast' disk drive that will plug into the machine's cartridge port to give speeds closer to normal disk drives. No less than five Commodore printers will work with the micro: a daisy wheel, two ordinary dot matrix printers, a colour dot matrix printer and a four-pen printer/plotter.

The Commodore Plus/4 is selling for approximately £70 more than the Commodore 64, but its improved BASIC and extra memory space make it a good buy. In a future article we shall be looking in detail at how this new version of Commodore BASIC works. Lack of software will remain a problem until the machine has established itself on the market. But a company with Commodore's record of sales should have little trouble gaining software support.

SEND IN THE CLONES

In the first instalment of this series, we took a preliminary look at Vu-Calc, a simple spreadsheet modelling package for the Sinclair Spectrum and BBC Micro. Here we discuss how to use Vu-Calc's features to carry out calculations of mortgage repayments or bank loans.

The great strength of spreadsheet programs — even a simple package like Psion's Vu-Calc — is the way in which they allow complex formulae to be applied to data. As we shall see, it is possible to build up some interesting and useful models with Vu-Calc, despite the fact that this particular program supplies almost nothing in the way of built-in formulae. In fact, Vu-Calc's sole built-in formula is its ability to 'sum' (i.e. add together the contents of) blocks of cells; this is indicated by prefixing the cell address with an @ sign.

More advanced spreadsheets contain very elaborate built-in formulae, which the user can call up by name. The advantage of such a system is that you don't actually have to know the mechanics of how these formulae work. If you want to use a mortgage formula with Multiplan, for example, to calculate expected repayments on a house purchase over different repayment periods — say 15, 20 and 25 years — you simply call up the formula and enter the relevant data. Multiplan then works out all the answers.

With Vu-Calc, the same calculations take considerably more time and effort. You must construct the required formulae yourself, and then key them into the machine. Vu-Calc also imposes a number of constraints on the user. It has a maximum of 28 columns, so the largest model you can build, with each column representing one month, will cover a period of just over two years. Accuracy can also be a problem — Vu-Calc works with integer (whole number) values only, and simply ignores the figures after a decimal point, so 99.9 would be considered as 99. Vu-Calc does allow values and arithmetical operations to be entered at any point on the model. For example, if the cursor is located at a blank cell (H5, for example), you can enter 500*2 in the command line. Pressing the Enter key will display the result — 1,000 — in cell H5.

Another irritating Vu-Calc feature is the way in which formulae are edited. A 'smart' package such as Lotus 1-2-3 uses a function key for editing. Pressing the key automatically puts the contents of the cell containing the cursor into the command line. Vu-Calc has an EDIT command (#E) that is used if a formula needs to be changed, but the formula must be retyped each time the edit facility is used. If you are working on a long formula and realise that you have forgotten to enter a bracket, there is no way of simply inserting it — instead the entire line must be retyped. All the EDIT command does is to tell the program to erase the old formula from a cell and then insert the new one.

However, using the REPLICATE command (#R) with a formula allows some fairly complex modelling to be done. Let's suppose that you want to extend the household budget example (see page 692) to anticipate inflationary increases in the household grocery budget, assuming a steady inflation rate of 0.5 per cent per month. Performing the necessary calculations with pencil and paper would clearly be a time-consuming task. With Vu-Calc it can be done quickly by using a formula and the REPLICATE command.

To carry out the desired operation, you must tell Vu-Calc to 'grow' your initial monthly budget (say £200) by 0.5 per cent. More sophisticated spreadsheets make this easy by using a GROW BY command, but Vu-Calc requires that the user enters the arithmetical operations that must be carried out. In order for Vu-Calc to recognise a formula that contains cell addresses, the formula must be prefixed with either $ or %. These are two arbitrarily chosen symbols that have nothing to do with dollars or percentages, but tell the program that cell addresses are significant in the formula under consideration, and these addresses are either relative (%) or absolute ($). An absolute cell reference tells Vu-Calc to look for and act on the value in a specific cell, regardless of that cell's position.

To see what a 'relative address' does, let us return to our example. The formula for 'growing' the budget by 0.5 per cent is %B3*100.5/100, where % indicates a relative cell address and B3 is the address of the cell containing the value representing the monthly food budget. Having keyed this formula into cell B4, we then need to copy the formula to get the result for the full year. B4 will display the numeric result of the formula — the formula itself appears at the bottom of the worksheet when the cursor is at B4. The REPLICATE command #R,B4,B5:B14 gives us the desired result (B4 contains the formula, B5:B14 defines the range of cells across which replication occurs). The results are shown almost instantly, and our spreadsheet model will look like this:

	1	2	3	4	5
A			JAN	FEB	MAR
B	Food budget		200	201	202

The display shows why cell B3 is used to hold the initial monthly budget amount — the label extends across columns one and two, so we start at column three in order to create a neat display. Note that all the figures are integer values — March's figure should read 202.005, but the spreadsheet 'rounds down', so this is displayed as 202 exactly. April's figure would really be 203.01502, but Vu-Calc would take it to be 203. As the inflationary increase grows larger month by month, so the discrepancy between the actual value and the figure displayed will also become greater.

This simple example demonstrates the effect of the REPLICATE command when used with relative cell addresses. Each time the program writes the formula into the next cell to the right, the formula changes accordingly. Our original formula in B4 was %B3*100.5/100. This formula replicates to B5 as %B4*100.5/100, to B6 as %B5*100.5/100, and so on. In each case the column number of the cell address is increased by one. Replicating down a column has the same effect on addresses (i.e. E1 becomes F1, etc.). If we had used absolute addresses ($) instead of relative addresses, this 'shift' in cell addresses would not have happened; instead the same formula would have been replicated across all the cells, and the value in each would be identical to the value shown in B4.

COST ANALYSIS

Now let's try using the same model to forecast a company's monthly expenditure on raw materials, starting at £100,000 per month and increasing by 0.5 per cent per month over two years. How much more would the goods cost if bought halfway through the second year? Using the model we have just built, this can be calculated very quickly.

Change the value in B3 to 100,000 by moving the cursor to B3 and typing in the new figure. Now use the REPLICATE command to extend the formula from B14 to B26 to make up the full 24 months. More sophisticated spreadsheets will show the new results the moment you change the value in B3. With Vu-Calc though, you must recalculate the results (which at the moment are still based on the old formula) by using the CALCULATE command, #C. Vu-Calc then calculates the new values and displays the answer we require in cell B20. If you try this example, you will find that the amount is £109,931 — an increase of nearly £10,000. This is not a precise figure, as all numbers are rounded down, but it is close enough to give you an idea of the effect of inflation over this period.

As a final example of the single-row type of problem, let's take a more complex formula, designed to work out the reducing balance of a £1,000 credit card debt or bank loan, on which interest is being paid at 27 per cent per annum. Assuming that you are paying back £80 per month, when can you expect to finish paying? The information needed to calculate this is the principal of the loan, plus the interest for the month, less the monthly repayment. So if we key in 1,000 in B1, the formula will be %B1+%B1*.27/12-80. Replicate the formula across all 28 columns in the model, scan the row to find the point at which the amount becomes positive, and you will have found the point at which the balance is paid off and you would be in credit if monthly payments continued. According to our model, this would take 16 months. As an added bonus, you also have a neat display of your outstanding balance each month, assuming you keep up the £80 payments.

In the next instalment, we will look at modelling on Abacus, the spreadsheet program offered with the Sinclair QL.

Relatively Absolute

In this simple model we have used the REPLICATE command across rows C, D and E; for the sake of example, we reproduce the formulae in each cell. Cell C3 has been absolutely replicated along the row, so the same formula, A5/12, appears in all the C cells, with the same result — £450. The formulae in cells D4 and E3, however, have been relatively replicated so the cell references in the formulae change from cell to cell along the row, with correspondingly variable results

	1	2	3	4	5	
A		ANNUAL INCOME =			5400	
B			JAN	FEB	MAR	
C	HOUSEHOLD INCOME		A5/12 450	A5/12 450	A5/12 450	
D	HOUSEHOLD EXPENSES		DATA 400	D3*1.01 404	D4*1.01 408	
E	MONTHLY BALANCE		C3-D3 50	C4-D4 46	C5-D5 42	

LIZ DIXON

CONTROLLING POWER

In this instalment of Workshop we look at the construction of a digital-to-analogue converter to add onto our user port system. The addition of this simple device will enable us to control analogue devices from the user port and produce digitally synthesised sound.

For this project we have opted to use a ready-made digital-to-analogue converter on a chip, although it is possible to build a circuit from discrete components. This reduces a complicated process to a relatively simple circuit.

The analogue output from this chip, the DAC chip, is buffered with an amplifier on a second chip. The output from this is fed directly to one output and through a capacitor and level control to the other output.

Step One: Cut the case to accommodate the two system bus connections. An outlet socket may also be cut for use in future projects.

Step Two: Cut the veroboard to size (30 holes by 16 strips). Now make the track cuts as shown in the diagram. Solder the chip sockets in place first, then the wire links. The two capacitors should be put in position next. It does not matter which way around they are fitted. If you wish to fit the bus extension socket, then solder this in place now and fit the ribbon cable.

Step Three: Fit the four sockets into the case, with the potentiometer. Make the connections between these with the tinned wire. Finally, make the three flying leads to the circuit board.

Step Four: Plug in the two chips and the converter is complete. Note that the two chips do not plug in the same way around: the D/A converter chip should be positioned so that the notch is to the left when viewed from above, with the male bus connector uppermost; the amplifier chip's notch should go to the right.

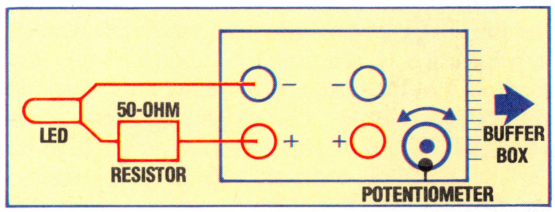
Once you have built the digital-to-analogue converter and carefully checked all the connections, you can test the unit. The D/A converter will convert any eight-bit binary value placed in the user port data register into a voltage. This voltage is output from the unit in two ways. At the DC (direct current) output socket pair, a DC voltage in the range 0 to +2.5v is obtainable, corresponding to digital user port values from 0 to 255. The other output pair is to enable us to simulate an AC (alternating current) output. The overall voltage level is controlled by a potentiometer and can be adjusted to suit the required input to another piece of apparatus.

In order to test the unit, we can devise a simple experiment to alter the brightness of an LED. To do this the following steps should be carried out:

Step One: An LED, of the type used in the original buffer box, (see page 523), should be connected in series to a 50-ohm resistor.

Step Two: The D/A converter unit should be connected directly to the buffer box, which in turn should be connected to the user port and have power supplied to it in the usual way.

Step Three: The LED and resistor circuit should be connected across the DC output sockets on the D/A converter box and this program run:



LIZ DIXON

```
 10 REM **** CBM64 D-TO-A TEST PROGRAM ****
 20 DDR=56579:DATREG=56577
 30 VL=127
 40 POKE DDR,255:REM ALL OUTPUT
 50 POKE DATREG,VL
 60 PRINT VL
 70 GET A$
 80 IFA$<>"Z" AND A$<>"X" THEN 70
 90 IF A$="X" THEN DV=1
100 IF A$="Z" THEN DV=-1
110 VL=VL+DV
120 IF VL<256 AND VL>=0 THEN 50
```

```
 10 REM **** BBC D-TO-A TEST PROGRAM ****
 20 DDR=&FE62::DATREG=&FE60
 25 value=127
 30 ?DDR=255:REM ALL OUTPUT
 40 REPEAT
 55 ?DATREG=value
 57 PRINTvalue
 60 A$=GET$
 62 IF A$<>"Z" AND A$<>"X" THEN 60
 65 IFA$="X" THEN dv=1 ELSE dv=-1
 67 value=value+dv
 70 UNTIL(value>255 OR value <0)
```

This simple program dedicates all eight user port lines to output by placing 255 (i.e. 11111111) in the data direction register. An initial value of 127 is then placed in the user port data register. By pressing the Z or X keys, the value in the data register is either decremented or incremented correspondingly. The program is terminated when the value in the data register falls outside the range 0 to 255.

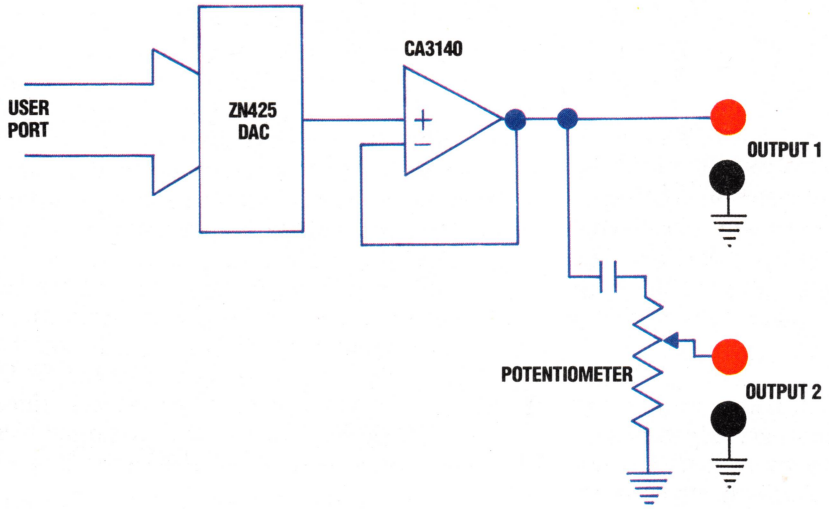
By increasing the *digital* value present in the data register we can produce increasing *analogue* voltages supplying the LED. As the voltage increases to an acceptable level, the LED will start to glow, dimly at first, and then more brightly as the voltage is further increased, until maximum brightness is achieved when the value 255 is present in the user port data register.

If your LED fails to light, try reversing the connections to the D/A converter box, before checking any further for possible faults. Unlike a normal bulb, which lights no matter which direction the current flows in, an LED will only light when the current is flowing in one particular direction.

Extract The Digit
The digital-to-analogue converter shown here features an excessively large potentiometer stalk. This is unnecessary, and should be trimmed to accommodate a control knob. Readers may well appreciate that in shopping for electronic components you must often take what you can get and make it fit


IAN McKINNELL

Full Circuit

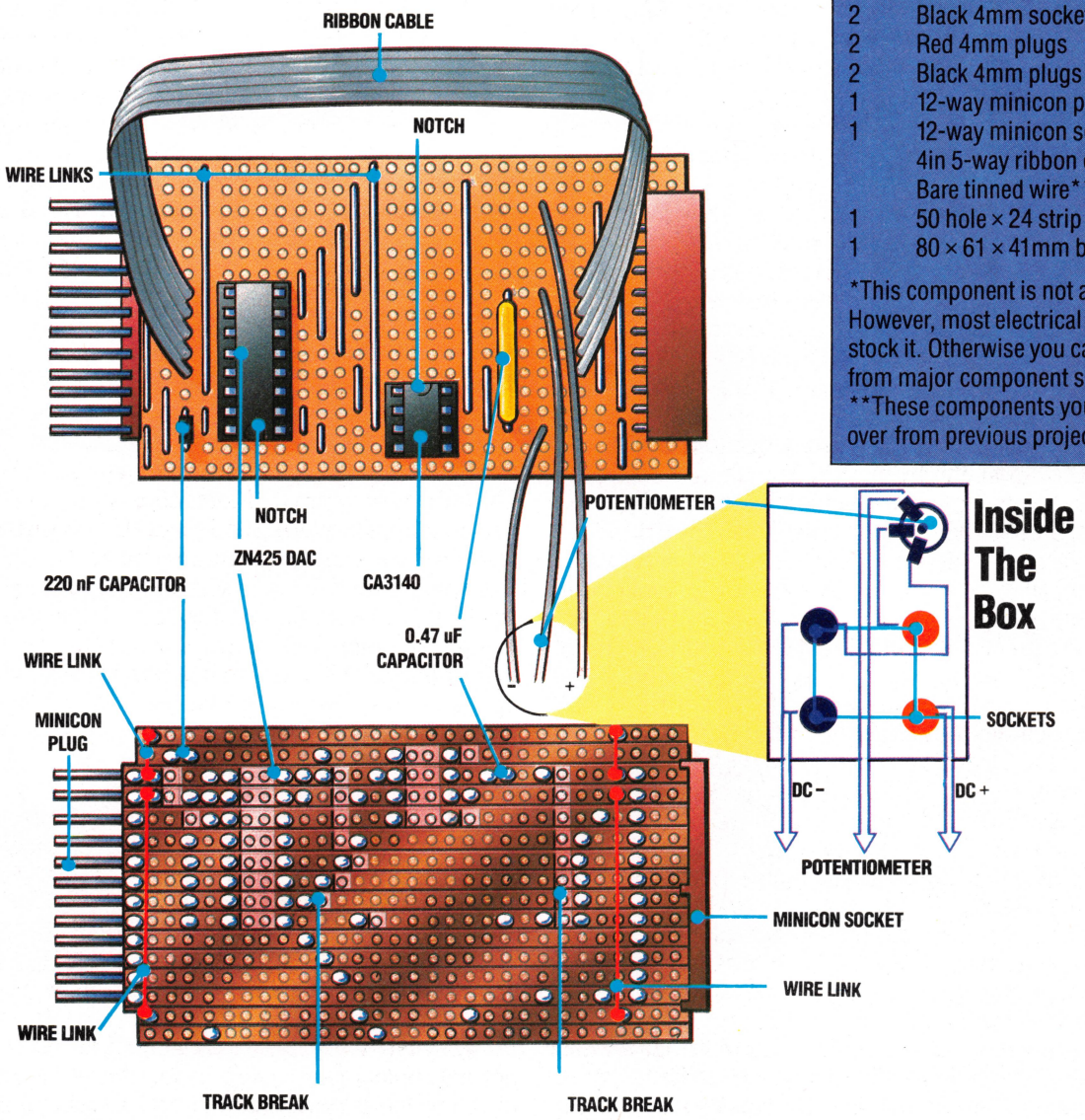


As usual in this sort of construction you must be scrupulously careful about the layout details, especially the location and cleanliness of the track breaks. Check your work regularly with the multimeter, insert passive components and wire links first, use solder and the soldering iron sparingly, and pay particular attention to positioning the chips

Parts List

No	Item	Maplin No
1	ZN425 DAC*	
1	CA3140 amplifier	QH29G
1	16-pin DIL socket	BL19V
1	8-pin DIL socket	BL17T
1	220nF capacitor	WW83E
1	0.47uF capacitor	BX80B
1	10K rotary potentiometer	FW02C
2	Red 4mm sockets	HF73Q
2	Black 4mm sockets	HF69A
2	Red 4mm plugs	HF66W
2	Black 4mm plugs	HF62S
1	12-way minicon plug	YW19V
1	12-way minicon socket	YW30H
	4in 5-way ribbon cable**	
	Bare tinned wire**	
1	50 hole × 24 strip veroboard	FL07H
1	80 × 61 × 41mm box	LH20W

*This component is not available from Maplin. However, most electrical component shops should stock it. Otherwise you can obtain one by mail order from major component suppliers.
**These components you should already have, left over from previous projects.



KEVIN JONES

GREEDY METHOD

Program algorithms can be written with either a strategical or a tactical approach. They may sometimes accept short-term deviations in the cause of long-term planning, or they may at every stage take the most direct route to the long-term goal. The latter method is called the *greedy method.* It has the failings that its name implies, namely wastefulness, lack of discrimination and the inability to see profit in apparent loss. If all the algorithms in a chess program, for example, were greedy, then essential subtleties such as position and sacrifice plays would be impossible; the program would play fast and aggressive chess but would be outranked easily by more pragmatic and carefully structured play.

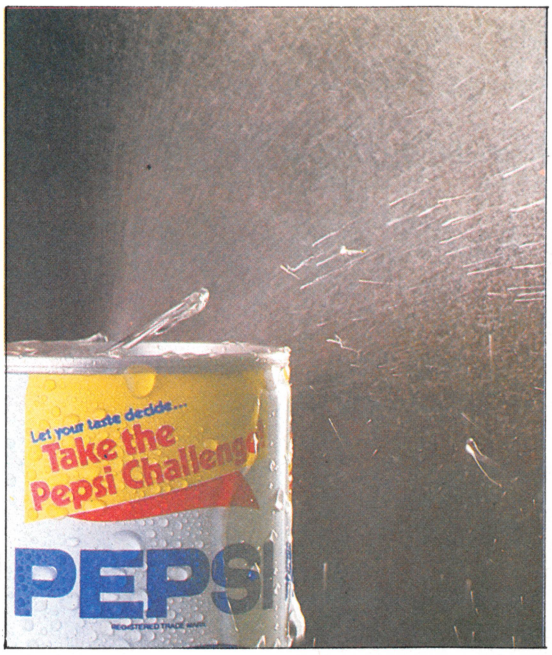
GROSCH'S LAW

Developed by H R J Grosch in 1953, *Grosch's Law* purports to give an indication of a computer system's profitability by using the formula:

$$\text{Performance} = (\text{Price})^2 \times (\text{A Constant})$$

During the 1950s this law was much quoted in the mainframe world, where all concerned stood to gain from encouraging the large centralised installations that the law suggested were cost-efficient — a system that costs three times as much as another should perform nine times better if this law is correct, although there was some debate as to whether the quantity (Price) should be squared or raised to a lesser power. The advent of integrated circuitry, however, has almost completely undermined the law.

Computer Crime
Massachusetts Institute Of Technology engineering students are reputed to hack into a Boston office block's control systems every year so that the lighted windows flash a giant message for an hour — but not all of it is so innocent: hackers allegedly caused the Pepsi-Cola Corporation's dispatching computer to divert shipments of Pepsi as a means of moving large sums of money into illicit accounts. Computer crime, especially using hacking methods, is reputedly growing at a faster rate than the computer industry itself



IAN McKINNELL

HACKING

In computing, *hacking* is a term applied to amateurs who devote considerable time and effort trying to crack software protection, customising operating systems, and breaking into other people's installations by using the telephone system. Hackers generally show a cavalier disregard for the concept of privacy, and in some cases are responsible for computer fraud — although it must be stressed that the majority of hackers are concerned only with testing their own limitations and those of the system they use. Hackers first came to the attention of the general public in the Walt Disney film *War Games*, in which an inspired amateur almost starts World War Three by breaking into the North American defence network.

HALF DUPLEX

A radio link as used by taxis, citizens' band and the emergency services is a *half duplex* connection — data can travel freely in both directions between stations but this cannot happen simultaneously, since one must be receiving while the other transmits, hence the need to say 'Over', or 'Come On Good Buddy for a big ten-four on that one' at the end of every speech. (See *full duplex* on page 676.)

HAMMING CODES

The transmission of data along wires inevitably introduces signal noise and errors, and computer scientists have developed many error-checking and error correction methods to counter data corruption. *Hamming codes*, invented by R W Hamming of Bell Telephone Laboratories in 1950, are a family of binary linear perfect error-correcting block codes, ideal for correcting any single error in the block.

Suppose we wish to send four bits of data in a block — 0111, for example. To them we add a three-bit Hamming code generated by the transmitting computer so that certain combinations of four bits from the seven will always contain an even number of ones. Here the code is 100, so the seven-bit block is 0111100 and the combinations are:

Data Code 0111 100		
Combinations	**Result**	**Logical Result**
0. 1. 1. 0	shows even number of ones	0
. 11. . 00	shows even number of ones	0
. . . 1100	shows even number of ones	0

The logical result of the three tests is 000, indicating no errors. Now suppose that in transmission bit four is 'flipped':

Data Code 0101 100		
Combinations	**Result**	**Logical Result**
0. 0. 1. 0	shows odd number of ones	1
. 10. . 00	shows odd number of ones	1
. . . 1100	shows even number of ones	0

The logical result of the three tests is 011, binary for three, which indicates that bit four — the third bit from the left of the block — has flipped, and so can be automatically corrected by the receiving computer. The tests fail only if more than one bit of the seven is corrupted, and for such cases there are BCH(Bose-Chandhuri- Hocquenghem)codes.

FREE TRANSFER

As the tutorial section of our 6809 Assembly language course draws to a close, we begin to take a more general look at the techniques of machine code programming. Our first topics are relocatable code, instruction lengths and timing routines.

A program that is written using *relocatable*, or *position-independent*, code can be placed at any position in memory and run without any changes having to be made. This is particularly important in multi-tasking or multi-user systems where several programs may be loaded into memory at the same time, and in order to ensure efficient use is made of memory space, the operating system must be able to load them at the most convenient place. Even in simpler, single-user systems it is usually important to be able to maintain subroutine libraries and to construct a program out of self-contained modules, in which case the position of a routine in memory may vary.

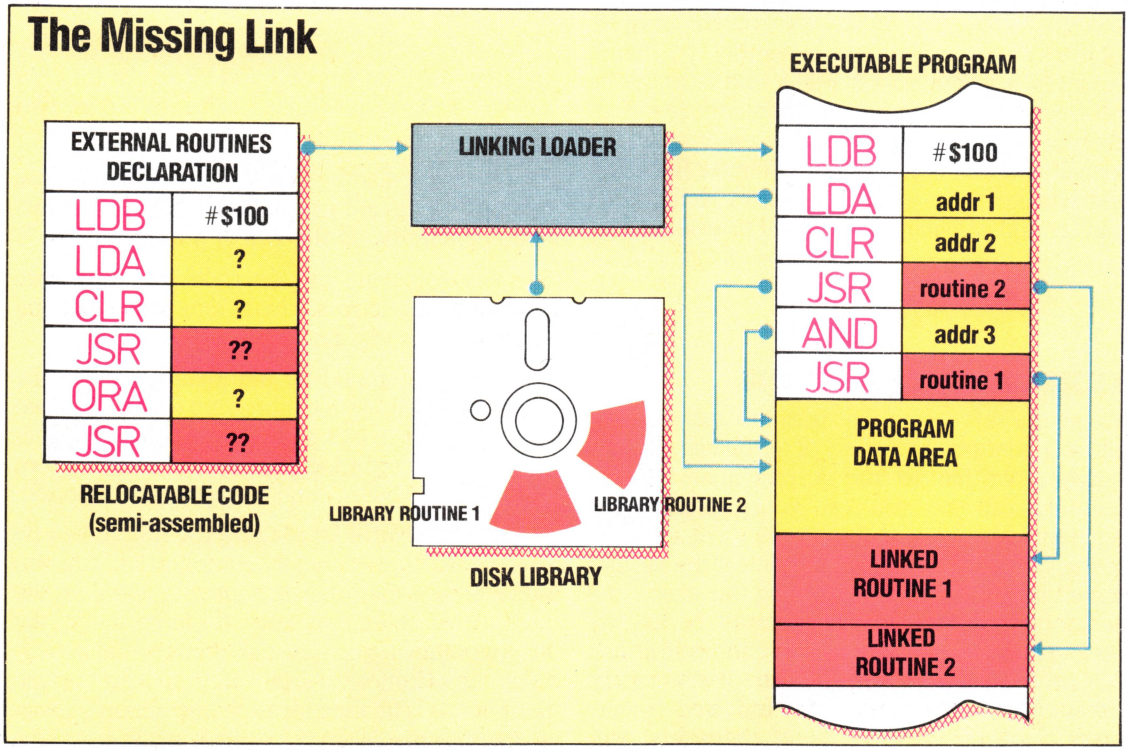
Most processors deal with this by using what is known as a *linking loader*. The assembler produces relocatable code, which leaves out all references to actual addresses in memory; it is the job of the linking loader to insert the addresses as it loads the program into memory. Since it is the loader itself that handles the addresses, it is straightforward to ensure that transfers of control between different modules are handled correctly. In this way, sections of code can be written in different languages that all compile or assemble to the same relocatable code; thus, for example, PASCAL programs can call FORTRAN library routines. This approach can also be used with the 6809, and indeed it is necessary if modular construction is used. The 6809 makes the process a lot easier by allowing fully relocatable code to be written directly, so there is no need for the extra stage of inserting addresses.

The key to writing relocatable code is to refer to all addresses by means of an *offset* from the program counter (PC). There are two ways in which a program can use an address: as data and as the destination for a transfer of control. Branch instructions (BRA, BSR, etc.) calculate their destinations as offsets from the PC and should be used for all transfers of control within the user program. The absolute transfer instructions (JMP and JSR) should be used only for destinations that will always be at the same place in memory, such as operating system routines.

The more difficult task is to make all the references to a data position independent, and the 6809 achieves this by allowing the PC to be used for indexing. The instruction:

```
LDA OFFSET,PC
```

will add the (signed) offset to the current value of



Linking Loader
In large systems, machine code programs are actually loaded into memory by the Linking Loader. This operating system utility takes the semi-assembled machine code (containing no absolute addresses) from the assembler, and determines the best ORG address for it from the current state of the system. It uses this address to replace the symbolic addresses that the assembler left in the program with absolute addresses, and then links to the program any library routines requested by the programmer; these routines are loaded from the library disk and attached to the program. Their absolute call addresses can then replace the symbolic addresses in the program. Finally, the Loader passes the complete program to the operating system for execution

the PC to obtain the *effective address*. The problem with this addressing mode is how to calculate the offset correctly: this requires calculating the difference between the address of the data and the current value of the PC, remembering that the PC is incremented as soon as an instruction is loaded into the processor. When the instruction is being executed, therefore, the PC points to the following instruction.

This method is complicated by the wide variation in the lengths of 6809 instructions — from one to five bytes long. For example:

```
LDX OFFSET,Y
```

takes up one byte for the op-code, and one byte for the *post-byte*, which is used for any indexed instruction to specify the index register being used, and whether or not indirection is to be taken into account. The offset can take up zero, one or two bytes depending on its size. Zero offsets and offsets that can be expressed in five bits can be incorporated in the post-byte (though some assemblers cannot handle the choice very accurately). Larger offsets require an extra byte (if they can be expressed in eight bits) or an extra two bytes. Special zero or five-bit offsets are not allowed when the PC is used for indexing. The instruction:

```
LDY OFFSET,X
```

would require yet another extra byte because the op-code for LDY is two bytes long.

If you enjoy writing Assembly language programs, and you are familiar with deciding what data addresses to use and where to locate your subroutines, then the associated tasks of looking up (and sometimes working out) the op-codes, converting addresses into two-byte format and manually calculating jump offsets will soon become second nature. A much simpler alternative to doing this assembly by hand, however, is to buy an assembler and let it do the work for you, since it must calculate the length of every instruction anyway. Most assemblers use the special notation PCR (Program Counter Relative), which makes the assembler use the PC as the index register and calculate the offset. For example:

```
DATITM  FCB  0
..............  ........
        LDA  DATITM,PCR
```

TERMINAL EMULATION

We give a subroutine that uses this technique to allow emulation of a variety of terminals, so that a program written to use a particular type of terminal can be run on your system. The differences between terminals are most apparent in the codes that are used to control the various screen functions, such as clearing the screen and positioning the cursor. These may be control codes (characters whose ASCII code is less than 32) or escape sequences, which consist of the Escape character (ASCII 27) followed by any other character or sequence of characters. Our simple routine allows only for the substitution of one control character by another, or a single character following Escape by another single character. But the routine clearly shows how such an emulation is carried out. Two tables are kept: one contains control characters; the other the Escape characters. If a program issues a control character, for example, then this character is used as an offset into the table to pick up the actual character that should be displayed.

Being fully relocatable, the routine can be added on to any other program in any position. We assume the existence of an operating system routine (OUTCH), which sends the character in the A accumulator to the screen, and we use JMP to access this routine — which should be at a fixed position in memory. Note that the ORG directive must still be given, although it has no effect. The character to be displayed should be in A.

INSTRUCTION LENGTHS

The problem of calculating the length of instructions is not confined to using 'program counter relative' addressing. It is often necessary to know the total length of a routine to be fitted into a restricted memory space — for example, in a ROM. Any book on 6809 Assembly language, or the manual of an assembler, should include a table of mnemonics along with associated data. For each mnemonic, this data would include a corresponding op-code, the total length of the instruction (though this may not be possible, in which case the minimum length will be given followed by a '+' sign), the number of clock cycles that the instruction takes to perform, and the effect of the instruction on the condition code flags.

The general rules for calculating the lengths of instructions — and hence for writing compact code — are:

1) Most op-codes are single byte; those directly affecting the contents of S and Y (except for LEA) and some affecting U (such as LDY and STS) are two bytes long.

2) Any indexed addressing will necessitate the use of a post-byte, and possibly a further one or two bytes depending on the size of the offset.

3) Data following the op-code for immediate mode will be one or two bytes long, depending on the size of the register used.

4) Addresses should be one byte long if in the direct page (usually locations zero to $FF), and two bytes otherwise. Not all assemblers make proper use of direct addressing, so an address may turn out to be two bytes when only one was expected.

The problem ot calculating the time taken by each instruction is equally complicated, if for no other reason than that the time depends on the number of bytes that must be fetched, i.e. the length of the instruction. This is important in real-time applications and for driving some peripherals. The time for each instruction is given as the number of clock cycles — or, at least, the

minimum number of clock cycles — that the instruction takes; so the actual time taken will also depend on the clock rate. A common clock rate for 6809 systems is one MHz — one million cycles per second. Thus, each clock cycle takes one millionth of a second. The straightforward instruction:

```
LDA DATITM
```

using a 16-bit address takes five cycles, and so executes in five millionths of a second. The instruction:

```
PSHS PC, B, CC
```

takes five cycles plus one cycle for each byte that is pushed onto the stack; in this case a total of nine cycles (remember that the PC is two bytes).

If a system does not include a real-time clock then the only way to measure elapsed time is by means of a software delay routine. This executes a sequence of instructions whose individual times have been chosen so that the sum gives the required interval. Such intervals are usually measured in milliseconds (thousandths of a second), so there is no need to be too exact — the odd millionth of a second will not matter. Assuming a clock rate of one MHz, the Software Delay routine we give here will produce delays in the range 1 to 255 milliseconds: the exact number of milliseconds (ms) being passed as a parameter in A. The notation (A) means the contents of accumulator A.

The calculation to find the constant COUNT can be expressed as follows:

Instruction	Number Of Clock Cycles	Number Of Times Executed	Time Taken (Clock Cycles)
PSHS B,CC	7	1	7
LDB #COUNT	2	(A)	(A) * 2
DECB	2	(A) * COUNT	(A) * COUNT * 2
BNE LOOP2	3	(A) * COUNT	(A) * COUNT * 3
DECA	2	(A)	(A) * 2
BNE LOOP1	3	(A)	(A) * 3
PULS PC,B,CC	9	1	9

This gives a total of (A) * (7 + 5 * COUNT) + 16 clock cycles. To make the calculation easier, we will ignore the 16. At a clock rate of one MHz, there are 1000 clock cycles in a millisecond so the total time should be (A) * 1000 clock cycles.

```
(A) * (7 + 5 * COUNT) = (A) * 1000
      (7 + 5 * COUNT) = 1000
            5 * COUNT = 973
                COUNT = 195 (to the nearest integer)
```

It is quite feasible to make more accurate delays, and to use the 16-bit registers for a greater range, but the principle of decrementing a register a fixed number of times remains the same.

Terminal Emulation Routine

```
ESCAPE  EQU   27
SPACE   EQU   32          (Space is ASCII 32)
OUTCH   EQU               Enter operating system
                          address here
        ORG   $1000
CTABLE  RMB   32          Table of control characters
ETABLE  RMB   128         Table of Escape characters
EFLAG   FCB   0           Flag to indicate whether last
                          character was an Escape
DISPCH  PSHS  X           Save X
        TST   EFLAG, PCR  Check if last character was an
                          Escape
        BEQ   DISP1       If not an Escape, then go to
                          DISP1
        LEAX  ETABLE, PCR Else get address of ETABLE in X
        LDA   A,X         Get replacement character using
                          the original character in A as the
                          offset
        CLR   EFLAG,PCR   Reset EFLAG
        BRA   FINISH
DISP1   CMPA  SPACE       Check if control character
        BGE   FINISH      If not control character, then
                          goto FINISH
        CMPA  ESCAPE      Else check if Escape
        BEQ   ESCCH       If it is Escape, then goto ESCCH
        LEAX  CTABLE,PCR  Get address of CTABLE in X
        LDA   A,X         Get replacement character using
                          the original character in A as
                          offset
        BRA   FINISH
ESCCH   INC   EFLAG,PCR   Set EFLAG to indicate character
                          was Escape
FINISH  PULS  X           Restore X
        JMP   OUTCH       Display character in A
        END               Note that the RTS at the end of
                          OUTCH will return control from
                          here to the calling program
```

Software Delay Routine

```
                          Subroutine to delay (A)
                          milliseconds
COUNT   EQU   195         See calculation
        ORG   $1000
DELAY   PSHS  B,CC        Save the other two registers
                          affected
LOOP1   LDB   #COUNT      Count for 1 ms
LOOP2   DECB              Keep decrementing
        BNE   LOOP2       Until B reaches zero
        DECA              Decrement A after each ms
        BNE   LOOP1       Until A reaches zero
        PULS  PC,B,CC     Return
```

PSYCHIC ATTACK

As the popularity of arcade style 'shoot-em-ups' has waned, so the distinction between the various types of computer game has become blurred. Best-selling home computer software now tends to combine elements of arcade, strategy and adventure games, and successful play requires much more than just fast reflexes.

Psytron is a complex and absorbing game featuring excellent fast-moving graphics, and will take the player a long time to master.

The player takes the role of the Psytron, a half-human, half-computer device that runs a space colony on the planet Betula 5. The colony is made up of various buildings, each of which has a specific function. Life-support systems are vital for the human colonists, and power plants are required to keep the computer operating. The most important installation is the main power plant, without which the entire colony would grind to a halt. By using either the keyboard or a joystick, the player is given a full 360° 'scan' of the installation, and each of the buildings is carefully drawn to give added realism to the view.

During early levels of play, Psytron behaves much like any other arcade game. The player has a number of weapons with which to fend off attacks by alien invaders, and it is not until the fourth level that the strategy elements of the game become apparent.

At the first level, the Psytron controls a 'droid', which is used to destroy the three-legged saboteurs that are 'teleported' into the base in an attempt to blow up the airlocks that connect the colony buildings. The second and third levels of play give the player the chance to shoot down the alien saucers — these may be picked off one by one or, if the 'Disruptor' is utilised, all aliens in view can be wiped out at once. However, the Disruptor is somewhat unstable, and there is a 10 per cent chance of it exploding when used.

Once level four has been reached, the player has the chance to make strategic decisions, based on the amount of damage suffered by the colony's installations. Throughout this level, the alien craft continue their attack, bombing strategically vital areas of the base and dropping saboteurs on kamikaze missions. However, at this level, 'Freezetime' is introduced. By simply pressing the Return key, the player may 'freeze' the action so that damage reports may be received and processed. Factors to be considered include the numbers of crew members who are dead or injured, the level of supplies remaining and the damage inflicted on the power plant. While in Freezetime, repairs may be carried out, and crew members can be allocated to areas in which they can be most effective. Careful juggling of resources is needed at this stage: the Psytron must take into account the fact that repair crews will consume more food and oxygen than non-working humans, and it may be necessary to abandon some of the buildings in order to conserve fuel, air and supplies.

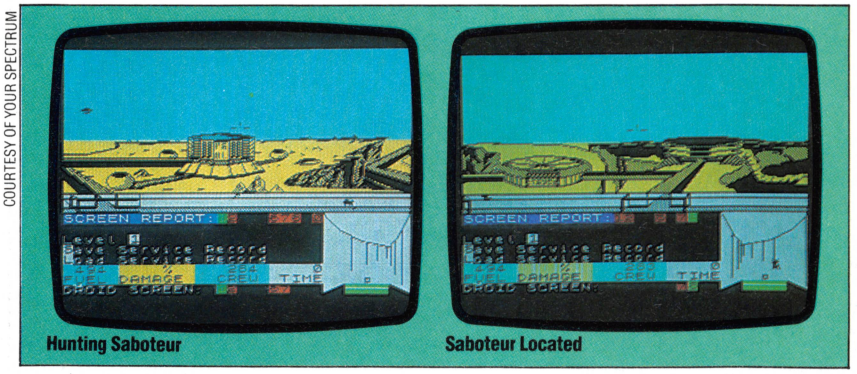
The fifth level gives the player the chance to communicate with a supply ship, which may beam vital supplies down to the colony. Care must therefore be taken to ensure that the docking facilities have not been damaged in previous attacks.

At the sixth and final level, the strategic factors assume paramount importance. The only goal at this stage is to survive for an hour, keeping the base intact by using all the facilities that have been introduced in previous levels. The number of attacking ships increases, the action speeds up, and it soon becomes apparent that it is impossible to keep all the colony's installations intact in the face of the overwhelming firepower of the attacking aliens. Decisions must be made as to which buildings must be sacrificed — it is vital to protect the docking bay, in particular, as this will allow stocks to be replenished.

Computer games have come a long way since the days of Space Invaders, and Psytron provides a demanding and absorbing alternative to the arcade games that have until recently dominated the market.

Droid's View
These scenes from Psytron show part of the action from level one of the game. As the droid chases the saboteur through the corridors, the view pans across the base. By means of the window in the bottom right-hand corner of the picture, the player is able to see through the eyes of the droid. When the saboteur appears in the window, the player can destroy it by pressing the fire button

COURTESY OF YOUR SPECTRUM



Hunting Saboteur — Saboteur Located

Psytron: For 48K Spectrum, £9.95, for Commodore 64, £7.95 (cassette); £11.95 (disk)
Publishers: Beyond Software, Competition House, Farndon Road, Market Harborough, Leics LE16 9NR
Authors: Tayo Olowu and Paul Voysey
Joysticks: Optional
Format: Cassette (Spectrum), cassette or disk (Commodore 64)

THE HOME COMPUTER ADVANCED COURSE

INDEX TO ISSUES 25 TO 36

THE HOME COMPUTER ADVANCED COURSE

INDEX TO ISSUES 25 TO 36